CS412/413

Introduction to Compilers and Translators
Spring '00

Lecture 4: Top-down parsing

Outline

• Eliminating ambiguity in CFGs
• Top-down parsing
• LL(1) grammars
• Transforming a grammar into LL form
• Recursive-descent parsing - parsing made simple

Where we are

Source code (character stream)

Lexical analysis

Token stream:

if b = 0 b = b;

Syntactic Analysis

Parsing/build AST

Abstract syntax tree (AST)

Semantic Analysis

Review of CFGs

• Context-free grammars can describe programming-language syntax
• Power of CFG needed to handle common PL constructs (e.g., parens)
• String is in language of a grammar if derivation from start symbol to string
• Ambiguous grammars a problem

Limits of CFGs

• Syntactic analysis can’t catch all “syntactic” errors
• Example: C++

```
HashTable<Key,Value> x;
```

• Need to know whether HashTable is the name of a type to understand syntax! Problem: “<”, “,” are overloaded
• Iota:

```
f(4)[f][2] = 0;
```

• Difficult to write grammar for LHS of assign – may be easier to allow all exprs, check later

if-then-else

• How to write a grammar for if stmts?

```
S → if (E) S
S → if (E) S else S
S → other
```

Is this grammar ok?
No—Ambiguous!

- How to parse?
  if \( (E) \) if \( (E) S \) else \( S \)

\[
S \rightarrow (E) \text{ if } \delta \text{ else } S
\]

Which “if” is the “else” attached to?

Grammar for Closest-if Rule

- Want to rule out \( (E) \) if \( (E) S \) else \( S \)
- Problem: unmatched if may not occur as the “then” (consequent) clause of a containing “if”

\[
\text{statement } \rightarrow \text{matched} | \text{unmatched}
\]

\[
\text{matched } \rightarrow \text{if } (E) \text{ if } (E) \text{ else matched}
\]

\[
\text{other } \rightarrow \text{if } (E) \text{ matched else unmatched}
\]

Top-down Parsing

- Grammars for top-down parsing
- Implementing a top-down parser (recursive descent parser)
- Generating an abstract syntax tree

Parsing a String Top-down

**Goal:** construct a leftmost derivation of string while reading in token stream

<table>
<thead>
<tr>
<th>Partly-derived String</th>
<th>Lookahead</th>
<th>Parsed part</th>
<th>Unparsed part</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( E )</td>
<td>( (1+2+(3+4)) ) ( +5 )</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow (E) S )</td>
<td>( 1 )</td>
<td>( (1+2+(3+4)) ) ( +5 )</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow (E+S) )</td>
<td>1</td>
<td>( (1+2+(3+4)) ) ( +5 )</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow (E+S) )</td>
<td>( 2 )</td>
<td>( (1+2+(3+4)) ) ( +5 )</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow (1+E+S) )</td>
<td>2</td>
<td>( (1+2+(3+4)) ) ( +5 )</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow (1+2+E) )</td>
<td>( 2 )</td>
<td>( (1+2+(3+4)) ) ( +5 )</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow (1+2+E+S) )</td>
<td>3</td>
<td>( (1+2+(3+4)) ) ( +5 )</td>
<td></td>
</tr>
</tbody>
</table>

Problem

- Want to decide which production to apply based on next symbol

\[
(1) \quad S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1)
\]

\[
(1)+2 \quad S \rightarrow E+S \rightarrow (S) + (E) + S \rightarrow (1) + E \rightarrow (1)+2
\]

- Why is this hard?

Grammar is Problem

- This grammar cannot be parsed top-down with only a single look-ahead symbol
- Not \( \text{LL}(1) \)
- Left-to-right-scanning, Left-most derivation, 1 look-ahead symbol
- Is it \( \text{LL}(k) \) for some \( k \)?
- Can rewrite grammar to allow top-down parsing: create \( \text{LL}(1) \) grammar for same language
Making an LL(1) grammar

- **Problem**: can’t decide which
  S production to apply until we see symbol after first
  expression

- **Left-factoring**: Factor common S prefix, add new non-terminal S’ at decision point. S’ derives (+E)*

- **Also**: convert left-recursion to right-recursion

Parsing with new grammar

\[
S \rightarrow E S' \\
S' \rightarrow S + S' | \epsilon \\
E \rightarrow number | ( S ) \\
\]

**Problem**: can’t decide which S production to apply until we see symbol after first expression.

**Left-factoring**: Factor common S prefix, add new non-terminal S’ at decision point. S’ derives (+E)*

**Also**: convert left-recursion to right-recursion

Predictive Parsing

- **LL(1) grammar**:
  - for a given non-terminal, the look-ahead symbol uniquely determines the production to apply
  - top-down parsing = predictive parsing
  - Driven by **predictive parsing table** of non-terminals × input symbols → productions

How to Implement?

- Table can be converted easily into a **recursive-descent parser**

Recursive-Descent Parser

```java
void parse_S () {
    switch (token) {
    case number: parse_E(); parse_S'(); return;
    case '(': parse_E(); parse_S'(); return;
    default: throw new ParseError();
    }
}
```

Using Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>E S'</td>
</tr>
<tr>
<td>S'</td>
<td>S + S'</td>
</tr>
<tr>
<td>E</td>
<td>number</td>
</tr>
</tbody>
</table>

EOF

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>E S'</td>
</tr>
<tr>
<td>S'</td>
<td>S + S'</td>
</tr>
<tr>
<td>E</td>
<td>number</td>
</tr>
</tbody>
</table>

lookahead token
Recursive-Descent Parser

```java
void parse_S() {
    switch (token) {
    case '+': token = input.read(); parse_S(); return;
    case ')': return;
    case EOF: return;
    default: throw new ParseError();
    }
}
```

Recursive-Descent Parser

```java
void parse_E() {
    switch (token) {
    case number: token = input.read(); return;
    case '(': token = input.read(); parse_S();
        if (token != ')') throw new ParseError();
        token = input.read(); return;
    default: throw new ParseError();
    }
}
```

Call Tree = Parse Tree

```
(1 + 2 + (3 + 4)) + 5

```

```
S   → ES'
S'  → +S → ε → ε
E   → number → (S)
```

How to Construct Parsing Tables

- Needed: algorithm for automatically generating a predictive parse table from a grammar

```
S → ES'  → ε  → ε
S' → +S  → ε  → ε
E → number → (S)
```

Parse Table Entries

- Consider a production X → γ
- Add → γ to the X row for each symbol in FIRST(γ)
- If γ can derive ε (γ is nullable), add → γ for each symbol in FOLLOW(X)
- Grammar is LL(1) if no conflicting entries
Computing nullable, FIRST

- X is nullable if it can derive the empty string:
  - if it derives ε directly
  - if it has a production $X \rightarrow YZ...$ where all RHS symbols (Y,Z) are nullable
  - Algorithm: assume all non-terminals non-nullable, apply rules repeatedly until no change in status

- Determining FIRST($\gamma$)
  - $\text{FIRST}(X) \supseteq \text{FIRST}(\gamma)$ if $X \rightarrow \gamma$
  - $\text{FIRST}(\alpha \beta) = \{ \alpha \}$
  - $\text{FIRST}(X \beta) \supseteq \text{FIRST}(X)$ if $X$ is nullable
  - Algorithm: Assume FIRST($\gamma$) = {} for all $\gamma$, apply rules repeatedly

Computing FOLLOW

- FOLLOW(S) $\supseteq \{ \$$ \}$
- If $X \rightarrow \alpha \beta$, FOLLOW(\gamma) $\supseteq$ FIRST(\beta)
- If $X \rightarrow \alpha \beta$ and $\beta$ is nullable (or non-existent), FOLLOW(\gamma) $\supseteq$ FOLLOW(X)
- Algorithm: Assume FOLLOW(X) = {} for all other X, apply rules repeatedly

- Common theme: iterative analysis. Start with initial assignment, apply rules until no change

Detecting ambiguity

- Construction of predictive parse table results in conflicts (but converse does not hold)

Completing the parser

Now we know how to construct a recursive-descent parser for an LL(1) grammar.

Can we use recursive descent to build an abstract syntax tree too?

Creating the AST

abstract class Expr {
  class Add extends Expr {
    Expr left, right;
    Add(Expr L, Expr R) { left = L; right = R; }
  }
  class Num extends Expr {
    int value;
    Num (int v) { value = v; }
  }
}
AST Representation

\[(1 + (2 + (3 + 4))) + 5\]

Creating the AST

- Just add code to each parsing routine to create the appropriate nodes!
- Works because parse tree and call tree have same shape
- `parse_S, parse_S', parse_E` all return an Expr:
  
  ```
  void parse_E() ⇒ Expr parse_E()
  void parse_S() ⇒ Expr parse_S()
  void parse_S'() ⇒ Expr parse_S'
  ```

AST creation code

```java
Expr parse_E() {
    switch(token) {
    case number: // E → number
        Expr result = Num(token.value);
        token = input.read(); return result;
    case '+' : // E → ( S )
        token = input.read();
        Expr result = parse_S();
        if (token != ')') throw new ParseError();
        token = input.read(); return result;
    default: throw new ParseError();
    }
}
```

```java
parse_S(Expr result, int left, int right) {
    if (right == 0) return left;
    else return left + right;
}
```

Or...an Interpreter!

```java
int parse_E() {
    switch(token) {
    case number:
        int result = token.value;
        token = input.read(); return result;
    case '(': // E → ( S )
        token = input.read();
        int result = parse_S();
        if (token != ')') throw new ParseError();
        token = input.read(); return result;
    default: throw new ParseError();
    }
}
```

```java
int parse_S(int left, int right) {
    if (right == 0) return left;
    else return left + right;
}
```

Summary

- We can build a recursive-descent parser for LL(1) grammars
  - Make parsing table from FIRST, FOLLOW
  - Translate to recursive-descent code
- Systematic approach avoids errors, detects ambiguities
- Next time: converting a grammar to LL(1) form, bottom-up parsing